Representing disparate resources by layering namespaces

Noah Evans
noah−e@is.naist.jp

ABSTRACT

Unix users have to deal with many different types of file formats, from widely varying character sets like shift jis or big5 to differing structured formats like xml and s−expressions. Users typically deal with these formats by manually converting one format to another using a unix tool like *tcs(1)* or *tgrep(1).*

This manual reformatting is tedious and against the unix philosophy of flexible tools dealing with one format in one character set.

Plan 9 and Inferno provide a solution to this problem by providing a standard character set, UTF−8, which can represent a wide variety of languages and by providing a way of arbitrarily representing file system data according to the needs of the user.

This allows the system programmer to deal with data formats on the file system level. By creating filesystems that do the conversion and detection of file formats automatically, users no longer need to manually convert foreign data formats, trusting the system to deal with foreign formats transparently.

Motivation

Acme has difficulty dealing with character sets other than UTF−8[Pike07]. Acme interprets any file it opens as UTF−8, treating all text as UTF−8 bytecodes. This makes it impossible to edit text from other character sets in acme, the text has already been converted to gibberish by acme’s attempt to render it in UTF−8.

this means that if a user has an unknown file the user must:

1. Open the file in acme.
2. Confirm that the file is not UTF−8.
3. Run *tcs(1)* to convert the file to the proper character set and save it.
4. Reopen the file in acme.

If this behavior is uncommon the process is relatively straightforward. However when dealing with a large number of files which potentially contain a variety of encodings, this process quickly becomes time consuming and difficult; the user must manually convert each file to UTF−8 to edit it in acme.

This manual conversion is antithetical to the philosophy of plan 9 and inferno, which emphasize a common representation for resources[Pike85][Presotto91].

Trfs

A good example of a way to deal foreign data formats is Nemo's `trfs` [Ballesteros02].

Inferno and Plan 9 users typically have difficulty opening files and traversing file systems imported from foreign operating systems. Files in operating systems like Windows and Mac OSX commonly have spaces in file names. Spaces make it impossible for tools that tokenize multiple files and directories on whitespace like *acme(1)* or the *plumber(4)* to open files. Trfs allows users to convert problem characters like spaces into characters that Plan 9 and Inferno can understand as parts of file names. This process of filename conversion occurs transparently and allows programs that adhere to Plan 9 and Inferno conventions to deal with filenames that would otherwise be unreadable.

Generalizing trfs

However `trfs` only deals with a small, limited problem —file names. An inferno or Plan 9 user trying to integrate Plan 9 with other operating systems still has to deal with a vast number of picture, file and character formats, each converted manually by the user or interpreted by a tool hardcoded to understand it.

A better way to deal with these formats would be to convert them to Inferno formats at the file system level like `trfs`.

Tmfs

I propose a file system called `tmfs(Transformfs)` which uses the same strategy as `trfs`, a filesystem that provides an alternate representation of the underlying namespace to tools. `Tmfs` differs from `trfs` in that it provides a framework for dealing with arbitrary functions. This framework is detailed in the following limbo adt:

```
Tranform: adt {
        clunk: fn(fid: int): int;
        remove: fn(fid: int): int;
        create: fn(fid: int,
                name: string,
                perm, mode: int): (int, string, int, int);
        open: fn(fid, mode: int): (int, int);
        read: fn(fid: int,offset: big,
                        count: int): (int, big, int);
        write: fn(fid: int, offset: big,
                        data: array of byte): int;
        walk: fn(fid, newfid: int,
                        names: array of string): (int, int);
        wstat: fn(fid: int): int;
};
```

This adt provides an interface to a module that allows the user to determine the translation they want to use on the file system. For instance `trfs` can be duplicated using `tmfs` with

```
Transform.create(fid: int,
                name: string,
                perm, mode: int): (int, string, int, int)
{
        name = tr(name, '#', ' ');
        return (fid, name, perm, mode);
}
```

with the rest of the module defined as identity functions and the tr function from the original *trfs(1).*

The user can then specify the translation necessary by loading a module and picking the insertion point in the namespace.

```
%tmfs -m trfs.dis ./
```

This provides a generalized flexible framework to deal with with a variety of different formats. Two potential uses of `tmfs` are detailed as follows:

Utfs

One obvious application of tmfs is solving the problem of opening non UTF-8 formatted data files in `acme`. If it were possible to detect and convert character sets at the file system level, the user would no longer need to exert any effort to deal with character sets in acme. Writes as well as reads would understand the necessary conversion, the original file would remain in its initial format keeping any changes made by acme.

The utfs module, currently under development, uses the mozilla character detection system[Li01] to determine the character set of a file during the open Tmsg. It then keeps a cache for every open fid. This cache determines which module from tcs to use when reading and writing to the underlying namespace. If the character set can not be determined the system does not alter the file data.

utfs is implemented as a simple shell script

```
#!/dis/sh

tmfs -m /lib/tmfs/utfs.dis $*
```

Tpfs

In addition to dealing with foreign character sets dealing with foreign structured data is much easier using tmfs. A proposed module tpfs uses freq to recognize structured data, tentatively xml and s-expressions. Structured data is then converted into tree paths, a way of describing structured data similar to unix paths. This data can then be processed using normal unix tools, eliminating the need to use specialized parsers or tools for a given format[Evans07]

Combining the two

However the real advantage of providing namespaces that interpret foreign formats is that the namespaces themselves can be layered to provide common interfaces to deal with complex and disparate formats. This process is similar to unix pipelines but it deals with conversions transparently. This transparency makes converting using namespaces easier for the user than doing ad hoc conversions with pipelines.

For example a common problem in natural language processing is extracting data from corpora, collections of linguistic data. This data is sometimes represented as structured text.

A hypothetical example would be trying to extract data from the British National Corpus and the Penn Chinese Treebank. The British National Corpus is in ascii so processing the characters is easy. However dealing with the Penn Chinese Treebank is more difficult because the data is in big5. If both corpora could be converted to UTF-8 that would greatly simplify processing. Both corpora could be processed using the same methods.

Utfs provides an ideal way to unify the character sets of both corpora. Given the large size of each corpus determining the character set using utfs would be easy.

However parts of the BNC and the Chinese Treebank are represented as structured text. The BNC represents structured text in xml; the Chinese Treebank represents data in s-expressions. To deal with both you have to handle both character set conversion and converting two dissimilar structured formats.

Tpfs provides the user with a consistent interface to the structured data. This allows to process the structured data in both corpora at the same time with the same tool chain.

Traditionally the user would have to solve these format problems by creating a shell script that would apply different conversion tools to each corpus directly during processing. Because conversion is done manually processing the corpus is cumbersome. The user must do extra work to determine the steps necessary to reach the desired format before figuring out what steps to take to extract the relevant information. Moreover this process of extraction is completely unrelated to the researchers end-goal, merely busywork.

Solving this problem is trivial using utfs and tpfs in tandem.

```
%cd corpora/
%utfs .
%tpfs .
%grep '.NP/' `{du -a *txt}
```

If the system administrator is aware of the locations of varying formats the administrator can configure the namespace in advance requiring no effort whatsoever from the user to deal with both corpora together.

Disadvantages

Character set detection still has its limitations. Especially for languages like Japanese which have a variety of similar character sets, determining the proper character set given a small file may be impossible. Worse the algorithm may detect the wrong character set, turning the file into into irrevocable gibberish. Right now utfs has no way of recovering from these sorts of errors. Future versions of tmfs will include a control file and method of interaction that allows the user to turn off interpretation for files whose character set cannot be determined.

Conclusions

Pipelines are still unix's most powerful abstraction. However using pipelines in ad hoc solutions to convert different character sets and structured formats to a common format is obnoxious and a waste of effort.

By encoding detection and conversion into a file system which interprets the current namespace in terms of a common format, the user can integrate varying formats and deal with them using common tools and idioms. This allows the user to concentrate on solving problems rather than figuring out how to support incompatible and inconsistent formats.

With tmfs as a foundation creating a variety of conversion systems becomes trivial. The user just defines modules to carry out the desired conversion. The user can even use the layered nature of the namespace to create a hierarchy of conversions similar to unix pipes but in a way that requires no effort from the user.

Acknowledgments

Francisco J Ballesteros for reviewing this paper and providing good concrete advice on how to improve this paper. Steve Simon for providing the idea of a file system for dealing with character sets. Rob Pike for explaining how acme deals with UTF-8. Eric Nichols for proofreading and advice. Yuuji Matsumoto and Masayuki Asahara for advice and support.

References

[Pike07] Pike, R., ‘‘Acme Edit/tcs and different character sets’’ *9fans mailing list*, http://groups.google.com/group/comp.os.plan9/msg/244b3f102cace624 2007

[Pike85] Pike, R. and Weinberger, P., ‘‘The Hideous Name’’ *USENIX Summer Conference Proceedings*, 1985.

[Presotto91] Presotto, D., Pike, R., Thompson, K. and Trickey, H., ‘‘Plan 9, A Distributed System’’ Proceedings of the Spring 1991 EurOpen Conference, Troms 1991

[Ballesteros02] F. J. Ballesteros, ‘‘trfs’’ *9fans mailing list*, http://groups.google.com/group/comp.os.plan9/msg/cf0151e48598d25e 2002

[Li01] Li, S. and Momoi, K., ‘‘A composite approach to language/encoding detection’’ *19th International Unicode Conference* 2001

[Evans07] Evans, N. and Asahara,M. and Matsumoto,Y., ‘‘Trees as paths: lessons from file systems and Unix in dealing with language trees’’, *IJCP NLP Proceedings 2007−NL−178−(9)* 2007